CSE 331
Software Design & Implementation

Kevin Zatloukal

Spring 2022

JavaScript & TypeScript

JavaScript & TypeScript

- Web apps running in the browser are usually not written in Java
  - it is possible to do so, but not typical

- JavaScript (JS) is the native language of web browsers
  - first-class functions
  - no compile-time types

- TypeScript adds compile-time types to JavaScript
  - important for correctness
  - competitors like Google's "Closure"

JavaScript (formally EcmaScript)

- Created in 1995 by Brendan Eich as a "scripting language" for Mozilla's browser
  - done in 10 days!

- No relation to Java other than trying to piggyback on all the Java hype at that time

- Often tricky due to its *simplicity*
  - example: no compile-time types
  - more examples later…

Playing with JavaScript

- Useful to play around on your own…
  - can't learn a language just from a talk

- Easy options:
  1. Use the "console" in Chrome
  2. Install and use "node" in the terminal

JavaScript console

Every browser has developer tools including the console, details about web pages and objects, etc.

A JS program can use `console.log("message");` to write a message to the console for debugging, recording, etc.

- "printf debugging" for JavaScript programs

In Chrome, right-click on a web page and select Inspect or pick View > Developer > Developer Tools from the menu. Click the console tab and you can see output that's been written there, plus you can enter JavaScript expressions and evaluate them. Super useful for trying things out.

Resources

- Lectures will (try to) point out key things
- For more: start with Mozilla (MDN) JavaScript tutorial:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript
- CodeAcademy has a good, free JavaScript basics course

- Be real careful about web searches – the JavaScript/ webapp ecosystem has way too many somewhat-to-totally incompatible or current vs. obsolete ways of doing similar things. Code snippets from the web may lead you *way* off.

Syntax

- Syntax like Java, C, C++, etc.
  - called the "C family" of languages
  - (includes C's predecessor B)

- `/*` comments `*/` or `//` comments

- Semicolons are optional at ends of lines and often omitted, but also encouraged ☺
  - "x = x + 1" allowed but "x = x + 1;" is better

Control flow – just like Java

- Conditionals

```
if (condition) {
 statements
} else if (condition){
 statements
} else {
 statements
}
```

- Loops

```
while (condition) {
 statements
}

for (init; condition; update){
 statements
}
```

  - Also for-of and for-in loops
    - Be careful with these. They have "interesting" semantics and differences that you need to get right if you use them.

Variables

- Variables have no type constraints:

```
let x = 3;
x = "ima string now!”;
```

- Introduced into program with **`let`**
  - use **`const`** for constants
  - older code uses **`var`** but **`let`** is preferred

Types

- Values do have (runtime) types, but just 6 of them:
  - number
  - string
  - boolean
  - null & undefined
  - Object (including arrays / lists)

- Has both null & undefined, which are similar
  - undefined is probably used more, means “not defined”
  - e.g., if no return statement, return value is undefined
  - e.g., if map doesn’t have key “a”, then map.get(a) should return undefined

Number Type

- All numbers are floating point! Even here:

```
for (let i = 0; i < 10; i++) { … }
```

- Usual numeric operations:
  - `+ - * /`
  - `++ --`
  - `+=`
  - ...

- Math methods (e.g., sqrt) much the same as in Java

String type

- Mostly the same as Java
  - immutable
  - most of the same methods as in Java
  - string concatenation with `+`
- But also string comparison with `<`
- Better string literals: `` `Hi, ${name}!` ``
  - `${name}` replaced by value of variable `name`

Boolean type

- Usual operators: **`&&`**, **`||`**, **`!`**

- But any value can be used in an "if"
  - "falsey" values: **`false`**, **`0`**, **`NaN`**, "", **`null`**, **`undefined`**
  - "truthy" values: everything else (including **`true`**!)

- Works… but a common source of bugs
  - more debugging…

Arrays

```
let empty = [ ]
let names = [ "bart", "lisa" ]
let stuff = [ "wookie", 17, false ]
```

- No type constraints, so types can be mixed
- Access elements with subscripts as usual
  - e.g., `stuff[0] // returns "wookie"`
- Methods like push, pop, shift, unshift, …
- Field length

Arrays

```
let stuff = [ "wookie", 17, false ]
stuff[4] = 331
console.log(stuff)
// ["wookie", 17, false, undefined, 331]
```

- No checking for array index out of bounds
  - automatically expand the array to have enough space
  - more debugging…

Objects

- Everything other than number, string, boolean, null, and undefined are Objects
  - mutable, *expandable* by default

- Simple syntax for creating them:

```
character = { name: "Lisa Simpson",
              age: 30 }
```

- Can retrieve fields in the usual way:

```
character.age = 7
```

Objects

- Really just a collection of name/value pairs
  - basically a HashMap (well, almost)

- Can also reference properties like this:

```
character["age"] = 7
```

  - like `character.get("age")` in Java

Objects

- Really just a collection of name/value pairs
  - basically a HashMap (well, almost)

- Can add & remove properties ("expandos"):

```
character.instrument = "horn"
delete character.age
```

- Not an error to retrieve a missing value
  - just returns undefined
  - more debugging…

Objects

- Quotes are optional in object literals:

```
let obj = {a: 1, "b": 2};
```

- But be careful:

```
let x = "foo";
console.log({x: x}); // {"x": "foo"}

let obj = {};
obj[x] = x;
console.log(obj); // {"foo": "foo"};
```

Equality

- Equality is complicated in any language

- JS has two versions: `===` (strict); `==` (loose)
  - `===`, `!==` check both types and values
  - `==` and `!=` can surprise you with conversions
    - `7 == "7"` is true!

- Object equality is reference equality
  - must compare contents yourself

Functions

- Named functions:

```
function average(x, y) {
  return (x + y) / 2;
}
```

- Anonymous, first-class (lambda) functions:

```
let f = function (x) { return x+1; }
let g = (x) => { return x+1; }
let h = (x, y) => (x + y) / 2;
```

more later…

Functions are values

- Functions are first-class values in JS
  - can be stored as values of variables, passed as parameters, and so on
  - lots of powerful techniques (see CSE 341)
    - we won't cover for the most part

```
let f = average;
let result = f(6, 7);    // 6.5
f = Math.max;
result = f(6,7);         // 7
```

Higher-level Functions

- Functions can be passed as parameters

```
function compute(f) {
  return f(2,3);
}

compute((a,b) => a+b);  // 5
compute((a,b) => a*b);  // 6
```

- See CSE 341 for more fun

Remember: no type constraints

```
function average(x, y) {
  return (x + y) / 2;
}
```

- No surprise

```
let result = average(6,7);     // 6.5
```

- But then…

```
let answer = average("6","7"); // 33.5!
answer = average(1,undefined); // NaN
```

Fake Classes

- JavaScript started as an OO language w/out classes
- Can do some of what we need already:

```
let obj = {f: (x) => x + 1};
console.log(obj.f(2));  // 3
```

- **Problem**: how would a method update the state (other fields) of the object?

`this`

- In the expression **`obj.method(...)`**:
  - **`obj`** is secretly passed to **`method`**
  - can be accessed using keyword **`this`**

- This works properly:

```
let obj = {
  a: 3,
  f: function (x) { return x + this.a }
};
console.log(obj.f(2));  // 5
```

Hazards of `this`

- Easy to write code that does not work:

```
let obj = {
    a: 3,
    f: function (x) { return x + this.a }
};
console.log(obj1.f(2));    // 5


let g = obj1.f;
console.log(g(2));         // NaN
```

Hazards of `this`

- Easy to write code that does not work:

```
let obj = {
    a: 3,
    f: function (x) { return x + this.a }
};

function compute(f) {
  return f(2);
}
console.log(compute(obj.f));    // NaN
```

Why should I care about `this`?

- We can add listener *functions* to components, but they don't know to pass this!

- This will not work inside of our class:

```
myMethod() {
  …
  btn.addEventListener("click", this.onClick)
}
```

- `this.onClick` produces a function, which it calls, but without the extra "`this`" parameter

How to fix `this`

- Inside of another method (where **`this`** is already set) the **`=>`** lambda syntax does this automatically:

```
myMethod() {
  …
  btn.addEventListener("click",
      () => this.onClick())
}
```

- The "**`this`**" inside of the **`=>`** expression means whatever **`this`** was in that method

How to fix `this`

- Alternative: create a function with **`this`** already set by using the **`bind`** method of a function object:

```
myMethod() {
  …
  btn.addEventListener("click",
      this.onClick.bind(this))
}
```

Classes

```
class Foo {
  constructor(val) {
    this.secretVal = val;
  }

  secretMethod(val) {
    return val + this.secretVal;
  }
}

let f = new Foo(3);
console.log(f.secretMethod(5));  // 8
```

Classes

- **`new Foo`** creates an object that has methods of the class
  - also calls the constructor

- Still has the same issue with this:

```
class Foo { … }

let f = new Foo(3);
let s = f.secretMethod;
console.log(s(5));                    // NaN

let t =  (x) => f.secretMethod(x);
console.log(t(5));                    // 8
```

JS vs Java Classes

- JS method signatures are just the name
  - JS objects are just HashMaps
  - field names are the keys

```
obj.avg(3, 5)
```

- Java methods signatures are name + arg types
  - e.g., `avg(int,int)`
- JS has only one method with a given name
  - language allows different numbers of arguments
    - missing arguments are undefined
  - can strengthen a spec by accepting a wider set of possible input types

Modules

- Each file is a separate unit ("namespace")
- Only exported names are visible outside:

```
export function average(x, y) { … }
```

- Others can import using:

```
import { average } from './filename';
```

  - file extension is not included